2do Parcial – 2018s2 - Estructuras de Datos – UNQ

Aclaraciones:

- *Esta evaluación es a libro abierto. Se pueden usar todas las funciones y tipos de datos vistos en la práctica y en la teórica, salvo que el enunciado indique lo contrario.*
- *No se olvide de poner nombre, nro. de hoja y cantidad total de hojas en cada una de las hojas.*
- *Se evaluará el entendimiento de la estructura, el cumplimiento de la eficiencia pedida en las operaciones, la correcta implementación de los algoritmos y el respeto de las barreras de abstracción.*

Random Access List

El objetivo de este parcial es implementar una lista de acceso aleatorio (Random Access List). Es similar a una lista, sólo que es posible acceder a un elemento en base a índice de forma eficiente. Y, en este caso, además es posible acceder al mínimo elemento, también de forma eficiente.

La representación que usaremos será la siguiente:

```
data RAList a = MkR Int (Map Int a) (Heap a)
```

En dicha representación se observa:

- Un `Int`, que representa la próxima posición a ocupar en la lista. Es decir, cuando se agregue un elemento al final, debe agregarse en dicha posición, que luego será incrementada. Cuando la estructura está vacía, el número es 0.
- Un `Map Int a`, que representa la relación entre índices (claves) y valores de la estructura.
- Una `Heap a` que contiene todos los valores de la estructura.

La interfaz a implementar, siendo N la cantidad de elementos, es la siguiente:

a) `emptyRAL :: RAList a`
Propósito: devuelve una lista vacía.
Eficiencia: $O(1)$.

b) `isEmptyRAL :: RAList a -> Bool`
Propósito: indica si la lista está vacía.
Eficiencia: $O(1)$.

c) `lengthRAL :: RAList a -> Int`
Propósito: devuelve la cantidad de elementos.
Eficiencia: $O(1)$.

d) `get :: Int -> RAList a -> a`
Propósito: devuelve el elemento en el índice dado.
Precondición: el índice debe existir.
Eficiencia: $O(\log N)$.

e) `minRAL :: Ord a => RAList a -> a`
Propósito: devuelve el mínimo elemento de la lista.
Precondición: la lista no está vacía.
Eficiencia: $O(1)$.

f) `add :: Ord a => a -> RAList a -> RAList a`
Propósito: agrega un elemento al final de la lista.
Eficiencia: $O(\log N)$.

g) `elems :: Ord a => RAList a -> [a]`
Propósito: transforma una `RAList` en una lista, respetando el orden de los elementos.
Eficiencia: $O(N \log N)$.

h) `remove :: Ord a => RAList a -> RAList a`
Propósito: elimina el último elemento de la lista.
Precondición: la lista no está vacía.
Eficiencia: $O(N \log N)$.

i) `set :: Ord a => Int -> a -> RAList a -> RAList a`
Propósito: reemplaza el elemento en la posición dada.
Precondición: el índice debe existir.
Eficiencia: $O(N \log N)$.

j) `addAt :: Ord a => Int -> a -> RAList a -> RAList a`
Propósito: agrega un elemento en la posición dada.
Precondición: el índice debe estar entre 0 y la longitud de la lista.
Observación: cada elemento en una posición posterior a la dada pasa a estar en su posición siguiente.
Eficiencia: $O(N \log N)$.
Sugerencia: definir una subtarea que corra los elementos del `Map` en una posición a partir de una posición dada. Pasar también como argumento la máxima posición posible.

Ejercicios

a) Definir los invariantes de representación en base a la estructura dada.

b) Implementar la interfaz usando dicha representación.

c) Definir la eficiencia de cada subtarea creada.

Interfaces

La interfaz de `Heap`, siendo H la cantidad de elementos en la heap:

`emptyH :: Heap a`	$O(1)$
`isEmptyH :: Heap a -> Bool`	$O(1)$
`findMin :: Heap a -> a`	$O(1)$
`insertH :: Ord a => a -> Heap a -> Heap a`	$O(\log H)$
`deleteMin :: Ord a => Heap a -> Heap a`	$O(\log H)$

La interfaz de `Map`, siendo K la cantidad de claves distintas en el map:

`emptyM :: Map k v`	$O(1)$
`assocM :: Ord k => k -> v -> Map k v -> Map k v`	$O(\log K)$
`lookupM :: Ord k => Map k v -> k -> Maybe v`	$O(\log K)$
`deleteM :: Ord k => k ->Map k v -> Map k v`	$O(\log K)$
`domM :: Map k v -> [k]`	$O(K)$